Parabola Volume 62 Issue 2 (2026), Article 2

The Scuffed Badminton Problem: finding the expected time of hitting state $(1, n)$ in a simplified scoring system

David Nachuan Chen[1]

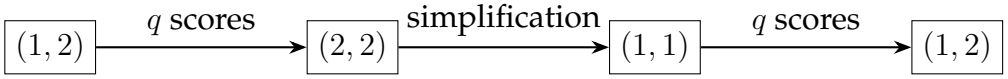
1 Introduction

Suppose that, in a badminton game of two players, the first and second player's score is a and b, respectively. We say that the score state is (a, b). In a classic badminton game, after each rally[2], the score increases by 1 which means that the next score state is either $(a + 1, b)$ or $(a, b + 1)$.

In this article, we examine the "scuffed badminton" scoring process: a two-player game in which the score (a, b) is continually reduced to $(a/\gcd(a, b), b/\gcd(a, b))$ after each rally. For example, a score state of $(3, 8)$ can either progress to $(4, 8)$ which simplifies to $(1, 2)$ or $(3, 9)$ which simplifies to $(1, 3)$. We shall see that such a scoring system drags a game on for much longer than it should have been, more evidently so if the two players are assumed to be equally skilled.

Specifically, since an identical score state can be achieved multiple times per game, we will examine only the expected number of rallies to reach a specific score state for the first time. In this article, I aim to explain the mathematical derivation of an algorithm that computes the expected hitting time (i.e., the number of rallies) to reach a score state of $(1, n)$, the resulting values obtained through the algorithm, and the experimental results obtained through running Monte Carlo simulations.

We will make some assumptions for the purposes of this analysis. Firstly, we will assume that both players are of equal skill level. This means from the score state (a, b), there is an equal probability of $\frac{1}{2}$ to progress to the score states $(a + 1, b)$ and $(a, b + 1)$ before possible simplification. Next, we assume that the initial state of the game is $(1, 1)$. Lastly, we consider the flipped score states to be identical. That is, score state (a, b) is equivalent to (b, a). We want to calculate the expected number of rallies for either player to reach $(1, n)$. Without loss of generality, we shall only consider the score states (a, b) for $a \leq b$. For example, for players p and q, it is possible that the score state $(1, 2)$ with p currently leading could become $(1, 2)$ again in the future, but with q leading, with the following sequence of events:



[1] David is a Computer Science/Advanced Maths major at UNSW and a maths Olympian.

[2] In badminton, a rally is a back-and-forth hitting of the shuttlecock between players until one player wins the exchange. I got the inspiration to invent this problem while playing badminton.

2 Methodology

Let E_n be the expected number of rallies to reach a score state of $(1, n)$ for the first time. Here, I will explain the ideas needed to derive the exact values of E_n.

Initially, $E_1 = 0$ as $(1, 1)$ is the starting position. Afterwards, the next score state will be $(1, 2)$ regardless of the player who wins the rally; hence, $E_2 = 1$. From this point onwards, we shall consider the expected number of rallies to progress from $(1, n)$ to $(1, n + 1)$ using Markov chains. In Figure 1, we show the Markov chain of score states up to $(1, 6)$, with each directed arrow indicating a possible move to the next score state. Furthermore, the directed arrows from $(1, n)$ to $(1, n + 1)$ also contain R, the expected number of rallies to progress to that state. The calculations of these expected values will be explained later.

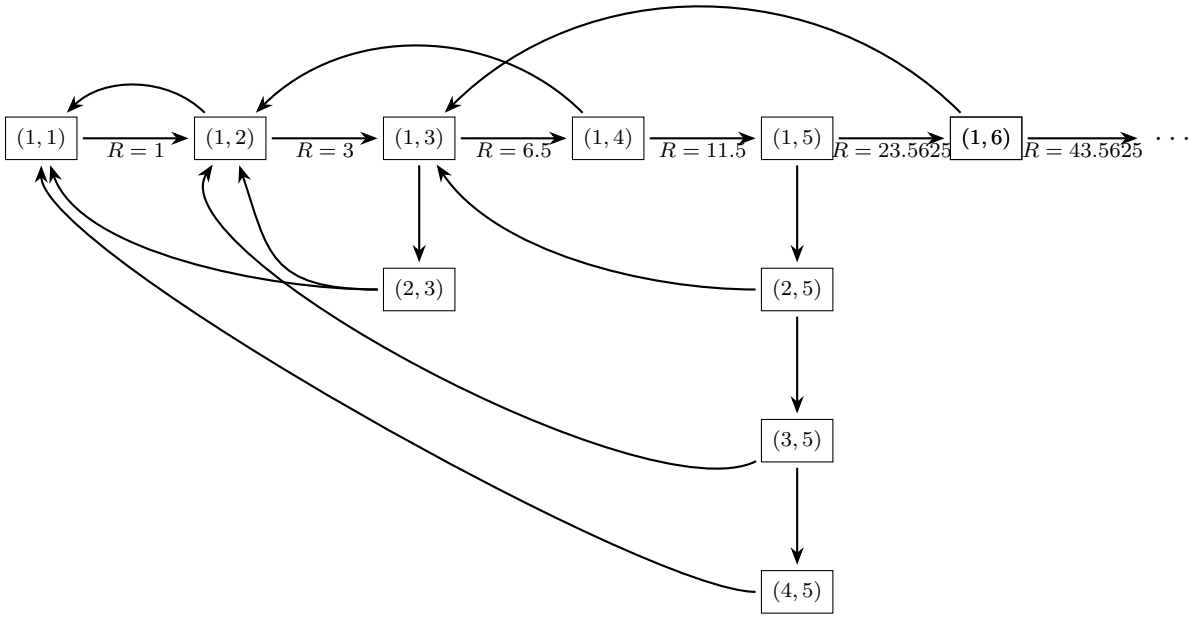


Figure 1: Markov chain of score states up to $(1, 6)$. All transition probabilities are $P = 0.5$, except for the arrow from $(1, 1)$ to $(1, 2)$ which is certain.

3 Algorithm

Now, we will mathematically derive the algorithm to compute the exact values of E_n.

Lemma 1. *Let h_n be the expected number of rallies to reach a score state of $(1, n)$ from $\big(2/\gcd(2, n), n/\gcd(2, n)\big)$ for $n \geq 2$. The expected number of rallies to reach $(1, n + 1)$ from $(1, n)$ is*

$$R_n = h_n + 2\,.$$

Proof. From the state score $(1, n)$, there is a probability of $\frac{1}{2}$ to advance to $(1, n+1)$ and a probability of $\frac{1}{2}$ to advance to $(a, b) = \big(2/\gcd(2, n), n/\gcd(2, n)\big)$. From (a, b), the expected number of rallies to return to $(1, n)$ is h_n.

We first prove that there is no sequence of events to reach $(1, m)$ from (a, b) for some $m > n$ before reaching $(1, n)$ again. Suppose the contrary. Here, we will examine the ratio y/x of the continually updated score states (x, y), initialised at (a, b), with $x \leq y$. The initial value of the ratio y/x is $n/2$, and we will observe this value until it becomes an integer. If x increases by 1, then the new ratio decreases whereas if y increases by 1, then the new ratio increases. However, I claim that, through these two operations, the ratio will always reach some integer value $k \leq \lceil \frac{n}{2} \rceil$ first. This is because the only ways to simplify down to an integer ratio $y/x = k$ are:

(a) x increases by 1 and becomes a factor of y.
Here, the previous value of y/x would be larger than k;

(b) y increases by 1 and becomes a multiple of x;

(c) $y/x = n/2$ is already an integer, given n is even.

If y increases by 1, but y/x is not an integer, then $\lceil \frac{y}{x} \rceil$ remains the same. Therefore, we conclude that the first integer value achieved by the ratio y/x must not be greater than $\lceil \frac{n}{2} \rceil$. Hence, the first score state $(1, k)$ that can be reached from any position $\big(2/\gcd(2, n), n/\gcd(n)\big)$ must always have $k \leq \lceil \frac{n}{2} \rceil < n$ for $n \geq 2$. Thus, it is impossible to advance to some score state $(1, m)$ from $\big(2/\gcd(2, n), n/\gcd(n)\big)$ without first advancing to $(1, n)$ for all $m > n$.

As such, there is a probability of $1/2$ that $h_n + 1$ rallies are taken to return to the state $(1, n)$. Then, there is a $1/2$ chance, upon returning to $(1, n)$, that the game progresses to $(1, n+1)$; hence, the probability of progressing to $(1, n+1)$ with an expected number of rallies of $h_n + 2$ is $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. Meanwhile, there is again a probability of $\frac{1}{2}$ that the game goes back to (a, b), thus repeating this process.

Here, we can see that the expected number of rallies to progress to $(1, n+1)$ becomes

$$\begin{aligned} R_n &= 1 \times \frac{1}{2} + (h_n + 2) \times \frac{1}{4} + (2h_n + 3) \times \frac{1}{8} + (3h_n + 4) \times \frac{1}{16} + \cdots \\ &= \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots \right) + \left((h_n + 1) \times \frac{1}{4} + 2(h_n + 1) \times \frac{1}{8} + 3(h_n + 1) \times \frac{1}{16} + \cdots \right) \\ &= 1 + \frac{h_n + 1}{4} \left(1 + 2 \times \frac{1}{2} + 3 \times \frac{1}{4} + \cdots \right). \end{aligned}$$

Using the identity

$$1 + 2x + 3x^2 + \cdots = (1 + x + x^2 + \cdots)^2 = \frac{1}{(1 - x)^2}$$

for $|x| < 1$, we deduce that

$$R_n = 1 + \frac{h_n + 1}{4} \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots \right)^2 = 1 + \frac{h_n + 1}{4} \times 2^2 = 1 + (h_n + 1) = h_n + 2 \,. \, \square$$

We will use Lemma 1 to design the algorithm that computes the values of E_n for $n \leq 11$.

Description

(a) Initialise the following: the array `ev` of values of $E_n = \texttt{ev}[n]$, the array **progress** of values of $R_n = E_{n+1} - E_n = \texttt{progress}[n]$, and the edges of the Markov chain using an adjacency list.

(b) Set $\texttt{ev}[1] = 0$, $\texttt{ev}[2] = 1$ and $\texttt{progress}[1] = 1$.

(c) For each $2 \leq n \leq 10$, we use breadth-first search (BFS) to find their respective values h_n. That is, starting from the state $(a, b) = \big(2/\gcd(2, n), n/\gcd(2, n)\big)$, the expected number of rallies to reach $(1, n)$. This can be done recursively using the function $\texttt{dist\_to\_n}(\texttt{int}\, i, \texttt{int}\, j, \texttt{int}\, n)$, where (i, j) is the current score state and $(1, n)$ is the desired score state. If $i \neq 1$, we define

$$\texttt{dist\_to\_n}(i, j, n) = 1 + \frac{\texttt{dist\_to\_n}(i_1, j_1, n) + \texttt{dist\_to\_n}(i_2, j_2, n)}{2}$$

where (i_1, j_1) and (i_2, j_2) are the immediate score states following (i, j). If $i = 1$, then we know that $j < n$. Hence, we know that

$$\texttt{dist\_to\_n}(1, j, n) = \sum_{k=j}^{n-1} \texttt{progress}[k]\,.$$

Here, we assert that all values of R_k up to $k = n - 1$ have been computed.

(d) Using Lemma 1, after computing $\texttt{progress}[n] = h_n + 2$ where $h_n = \texttt{dist\_to\_n}(a, b, n)$, we set

$$\texttt{ev}[n + 1] = \texttt{ev}[n] + \texttt{progress}[n]\,.$$

The efficacy of the algorithm ends in the score state $(1, 11)$. This is because $(2, 11)$ can "escape" to higher score states such as $(16, 17)$, in such a way that the algorithm terminates at a segmentation fault due to either of the following factors:

(a) Going out of bounds of the Markov chain. Not even large Markov chains, consisting of score states up to $(\sim 9000, \sim 9000)$, are able to contain the possible journeys beginning from $(2, 11)$. Any Markov chain of larger size runs into memory allocation issues on my machine.

(b) Stack overflow. I implemented a depth-first search (DFS), originating from $(2, 11)$, that terminates whenever a score state is reached that has been previously reached. Furthermore, I defensively implemented a check for in case the search may go out of bounds. Here, the maximum score state is set to $(9000, 9000)$. However, my machine still ran out of memory before the search was completed.

Hence, it is very difficult to obtain the exact values of E_n algorithmically for $n \geq 12$, a Monte Carlo simulation would be appropriate to estimate such values.

4 Results

Here, we examine the results obtained by running the above algorithm, as well as a Monte Carlo simulation of this game.

n	E_n (decimal)	E_n (fraction)	$\ln(E_n + 1)$	$E_n/2^n$
1	0	0	0.000	0.000
2	1	1	0.693	0.250
3	4	4	1.609	0.500
4	10.5	21/2	2.442	0.656
5	22	22	3.135	0.688
6	45.5625	729/16	3.841	0.712
7	89.125	713/8	4.501	0.696
8	176.62109375	45215/256	5.180	0.690
9	344.7421875	44127/128	5.864	0.673
10	679.484375	43487/64	6.523	0.664
11	1338.96875	42847/32	7.200	0.654

Table 1: Values of E_n in decimal and fractional form, their logarithms, and ratio to 2^n.

In Figure 2, we plot the values $\ln(E_n + 1)$ against n for $1 \leq n \leq 11$. Using basic statistical tools, it can be inferred that the line of best fit (shown in red) is $y = 0.7170x - 0.5774$. Furthermore, the coefficient of determination is $R^2 = 0.9978$.

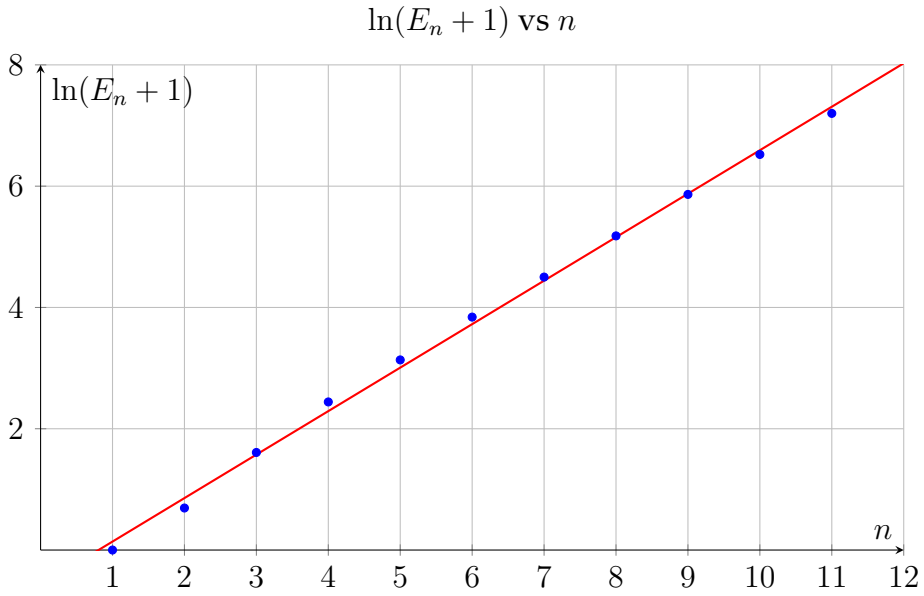


Figure 2: Plot of $\ln(E_n + 1)$ against n with the line of best fit.

My friend Mingzhou Lee[3] was kind enough to help me implement a Monte Carlo

[3]Mingzhou is a UNSW student majoring in Computer Science/Advanced Maths. He plays badminton

simulation to collect experimental results for $n \geq 12$, as well as to verify the algorithmically computed results I obtained. His code is listed in the Appendix.

n	$\overline{E_n}$	Mean Runtime (μs)	Difference from E_n (%)
1	0.0000	0.09	0.0000
2	1.0000	0.13	0.0000
3	3.9981	0.22	0.0475
4	10.5032	0.41	0.0305
5	22.0221	0.71	0.1005
6	45.5104	1.40	0.1143
7	89.2854	2.68	0.1800
8	176.7809	5.23	0.0905
9	344.4500	11.08	0.0848
10	680.4348	23.69	0.1399
11	1337.8038	44.28	0.0870

Table 2: The runtime and accuracy of computed E_n compared to actual values, 1000000 trials each.

From the table above, we see that the Monte Carlo simulation validates my algorithm, as the experimental values differ by less than 0.2% from my computed exact values. Furthermore, the values $\overline{E_n}$ of computed E_n are given for $n \in \{12, 13, 14, 15\}$ along with a 95% confidence interval for the values of E_n.

n	$\overline{E_n}$	**Mean Runtime**	**No. of Trials**	**95% Confidence Interval**	$\overline{E_n}/2^n$
12	2656.1969	85.89 μs	1000000	$[2651.0068, 2661.3870]$	0.648
13	5257.6528	164.36 μs	500000	$[5243.0922, 5272.2135]$	0.642
14	10481.2773	379.75 μs	200000	$[10435.3185, 10527.2362]$	0.640
15	20942.7983	725.00 μs	100000	$[20812.7248, 21072.8718]$	0.639

Table 3: Runtime of computed E_n for $n = 12, 13, 14, 15$ with 95% confidence interval.

5 Conclusion

From the experimental results, we can conclude that the values of $E_n \ll 2^n$. Specifically, $E_n \approx 0.65 \times 2^n$ for $n \leq 15$. However, Monte Carlo simulations for higher values of n exhibit a greater variance than simulations for lower values of n, thus estimating E_n becomes increasingly difficult.

Furthermore, the results indicate that $E_{n+1} < 2E_n$ for $n \geq 6$, and therefore it is likely that the ratio $E_n/2^n$ will continue to decrease as exhibited. At least for even values of $n \geq 6$, this "almost-doubling" effect follows from Lemma 1. For even values of n, we

with me and is quite good at it.

see that $h_n = E_n - E_{n/2}$, as that is the expected hitting time from $(1, n/2)$ to $(1, n)$. Since by Lemma 1 we have that $E_{n+1} = E_n + h_n + 2$, we obtain

$$E_{n+1} = E_n + (E_n - E_{n/2}) + 2 = 2E_n - E_{n/2} + 2\,.$$

For even $n \geq 6$, $E_{n/2} > 2$, and thus $E_{n+1} < 2E_n$.

The doubling nature of E_n can also be intuitively explained. Whenever you reach the state $(1, n)$, there is a 50% chance that you can immediately proceed to $(1, n+1)$. Essentially, this means that it takes approximately twice as long to reach $(1, n+1)$, as it will take to reach $(1, n)$.

6 Further discussion

We discuss how this problem relates to other fields in maths and how the ideas employed here can translate into variations of this problem.

Although the exact formulation of this problem has never been explored, it is a variant of random walks and Markov processes. In addition, the frequency of score simplifications is proportional to the density of numbers that are co-prime to n, thus relating it to number theory and Euler's totient-phi function.

This article presents the most basic form of this problem: two evenly-matched players facing off without regard of the order of the scores. A natural extension to this problem would be a scenario in which the two players are unevenly matched, which can be approached similarly using Markov chains. Another consideration would be to play this game with more than two players with varying skill levels, adding an extra dimension to this problem. However, a more immediate extension of this problem would be to consider the expected hitting time to visit any given score state for the first time. The main challenge to this problem is that there exist infinitely many possible routes to reach a state such as $(2, 3)$ for the first time. For example, it is possible to reach a state such as $(2, 5)$ before reaching $(2, 3)$, with the following sequence:

$$(1,1) \to (1,2) \to (1,3) \to (1,4) \to (1,5) \to (2,5) \to (1,3) \to (2,3)\,.$$

Thus, it becomes much more difficult to calculate the exact expected hitting time for score states not in the form $(1, n)$. However, I believe that Monte Carlo simulations are a viable experimental approach for many extensions of this problem, and we take these results as baseline estimates for the actual true values of this problem.

Acknowledgements

I want to first and foremost offer my sincerest thanks to Dr Thomas Britz for proofreading my paper. I cannot thank him enough for his invaluable feedback. I would also like to thank my friend Mingzhou Lee for his contributions in developing the Monte Carlo simulation for this problem.

Appendix

Algorithmic Implementation in C++

This algorithm calculates the exact value of E_n for the values $n \leq 11$, using Markov chains. This is developed by the author.

```
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;

double ev[100];
vector< pair<int, int> > markov_chain[100][100];
int g;
double progress[100];
double ans;

int gcd(int a, int b) {
    if (a == b) {
        return a;
    } else if (a == 0) {
        return b;
    } else if (b == 0) {
        return a;
    } else if (a > b) {
        return gcd(a % b, b);
    } else {
        return gcd(a, b % a);
    }
}

void construct_chain() {
    for (int i = 0; i < 99; i++) {
        for (int j = i; j < 99; j++) {
            g = gcd(i + 1, j);
            markov_chain[i][j].emplace_back((i + 1) / g, j / g);

            g = gcd(i, j + 1);
            markov_chain[i][j].emplace_back(i / g, (j + 1) / g);
        }
    }
}

double dist_to_k(int i, int j, int k) {
    if (i == 1) {
        ans = 0;
        while (j < k) {
            ans += progress[j];
            j++;
        }
        return ans;
```

```
    } else {
        return 1 + dist_to_k((markov_chain[i][j])[0].first, (markov_chain[i][j])[0].second, k) / 2 + dist_to_k((markov_chain[i][j])[1].first, (markov_chain[i][j])[1].second, k) / 2;
    }
}

int main() {
    ev[1] = 0;
    ev[2] = 1;
    progress[1] = 1;

    construct_chain();

    for (int j = 2; j <= 10; j++) {
        progress[j] = dist_to_k((markov_chain[1][j])[0].first, (markov_chain[1][j])[0].second, j) + 2;
        ev[j + 1] = ev[j] + progress[j];
    }

    for (int j = 1; j <= 11; j++) {
        cout << ``E_" << j << ``: `` << fixed << setprecision(8) << ev[j] << '\n';
    }
}
```

Depth-first search from $(2, 11)$ in C++

This DFS traversal demonstrates why the previous algorithm would not be effective for calculating exact values of E_n for $n \geq 12$. This is implemented by the author.

```
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;

const int MAX = 9000;

int g;
vector<vector<bool>> visited(MAX, vector<bool>(MAX));

int gcd(int a, int b) {
    if (a == b) {
        return a;
    } else if (a == 0) {
        return b;
    } else if (b == 0) {
        return a;
    } else if (a > b) {
        return gcd(a % b, b);
```

```
    } else {
        return gcd(a, b % a);
    }
}

void dfs_visit(int i, int j) {
    cout << i << ' ' << j << '\n';
    visited[i][j] = true;
    if (j == MAX - 1) {
        return;
    } else if (i == 1 || j == 1) {
        return;
    } else {
        g = gcd(i + 1, j);
        if (!visited[(i + 1) / g][j / g]) {
            dfs_visit((i + 1) / g, j / g);
        }

        g = gcd(i, j + 1);
        if (!visited[i / g][(j + 1) / g]) {
            dfs_visit(i / g, (j + 1) / g);
        }
    }
}

int main() {
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            visited[i][j] = false;
        }
    }

    // Begin DFS traversal from (2, 11)
    dfs_visit(2, 11);
}
```

Monte Carlo implementation in Python

This code uses multi-threading to efficiently run a simulation of the game for hundreds of thousands of trials. This is developed by the author and Mingzhou Lee.

```
import random
import time
import math
from concurrent.futures import ProcessPoolExecutor

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

```
10
11 def simplify_score(a, b):
12     g = gcd(a, b)
13     return (a // g, b // g)
14
15 def simulate_game_true_rules(winning_score_n, max_rounds=1_000_000):
16     current_score = (1, 1)
17     rounds = 0
18     while rounds < max_rounds:
19         if winning_score_n in current_score:
20             if (current_score[0] == winning_score_n and current_score[1]
       == 1) or \
21                 (current_score[0] == 1 and current_score[1] ==
      winning_score_n):
22                 break
23         rounds += 1
24         if random.random() < 0.5:
25             new_score = (current_score[0] + 1, current_score[1])
26         else:
27             new_score = (current_score[0], current_score[1] + 1)
28         current_score = simplify_score(*new_score)
29     return rounds
30
31 def worker(trials, n):
32     return [simulate_game_true_rules(n) for _ in range(trials)]
33
34 def monte_carlo_parallel(winning_score_n, num_trials=100000, workers=
      None):
35     start = time.time()
36     if workers is None:
37         import os
38         workers = os.cpu_count() or 4
39
40     # split trials evenly across workers
41     chunk = num_trials // workers
42     chunks = [chunk] * workers
43     chunks[-1] += num_trials % workers  # handle remainder
44
45     with ProcessPoolExecutor(max_workers=workers) as executor:
46         results = executor.map(worker, chunks, [winning_score_n] *
      workers)
47
48     # flatten results
49     all_rounds = [r for chunk in results for r in chunk]
50     avg = sum(all_rounds) / len(all_rounds)
51
52     # compute standard deviation
53     mean = avg
54     variance = sum((x - mean) ** 2 for x in all_rounds) / (len(
      all_rounds) - 1)
55     std_dev = math.sqrt(variance)
56
57     # compute 95% confidence interval
```

```
    z = 1.96  # 95% CI
    margin_of_error = z * (std_dev / math.sqrt(len(all_rounds)))
    ci_lower = mean - margin_of_error
    ci_upper = mean + margin_of_error

    elapsed = time.time() - start
    print(f"Completed {num_trials} trials in {elapsed:.2f} sec using {workers} workers.")
    print(f"Average game length: {avg:.4f}")
    print(f"95% CI: [{ci_lower:.4f}, {ci_upper:.4f}]")
    return avg, (ci_lower, ci_upper)

if __name__ == ''__main__":
    # change these values
    WINNING_SCORE_N = 15
    NUM_TRIALS = 100000
    monte_carlo_parallel(WINNING_SCORE_N, NUM_TRIALS, workers=8)
```